

11/12/03

EXHIBIT A

Intel Architecture Software Developer's Manual

Volume 1: Basic Architecture

NOTE: The *Intel Architecture Software Developer's Manual* consists of three books: *Basic Architecture*, Order Number 243190; *Instruction Set Reference Manual*, Order Number 243191; and the *System Programming Guide*, Order Number 243192.

Please refer to all three volumes when evaluating your design needs.

1997

A thick, solid black vertical bar located at the bottom right of the page, below the year 1997.

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Intel's Intel Architecture processors (e.g., Pentium® processor, Pentium processor with MMX™ technology, Pentium Pro processor, and Pentium II processor) may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation
P.O. Box 7641
Mt. Prospect IL 60056-7641

or call 1-800-879-4683
or visit Intel's website at <http://www.intel.com>

Copyright © Intel Corporation 1996, 1997.

* Third-party brands and names are the property of their respective owners.

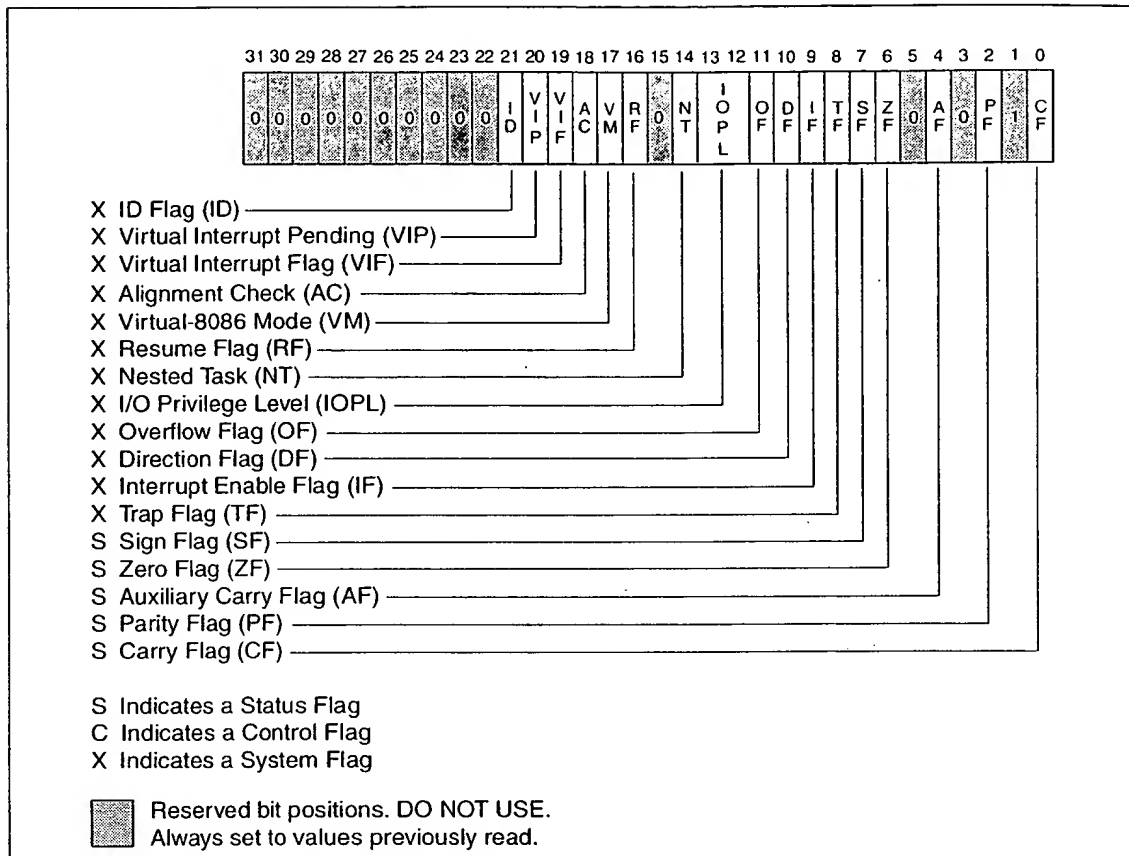


Figure 3-7. EFLAGS Register

As the Intel Architecture has evolved, flags have been added to the EFLAGS register, but the function and placement of existing flags have remained the same from one family of the Intel Architecture processors to the next. As a result, code that accesses or modifies these flags for one family of Intel Architecture processors works as expected when run on later families of processors.

3.6.3.1. STATUS FLAGS

The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, MUL, and DIV instructions. The functions of the status flags are as follows:

CF (bit 0)

Carry flag. Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.

PF (bit 2)	Parity flag. Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.
AF (bit 4)	Adjust flag. Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.
ZF (bit 6)	Zero flag. Set if the result is zero; cleared otherwise.
SF (bit 7)	Sign flag. Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)
OF (bit 11)	Overflow flag. Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.

Of these status flags, only the CF flag can be modified directly, using the STC, CLC, and CMC instructions. Also the bit instructions (BT, BTS, BTR, and BTC) copy a specified bit into the CF flag.

The status flags allow a single arithmetic operation to produce results for three different data types: unsigned integers, signed integers, and BCD integers. If the result of an arithmetic operation is treated as an unsigned integer, the CF flag indicates an out-of-range condition (carry or a borrow); if treated as a signed integer (two's complement number), the OF flag indicates a carry or borrow; and if treated as a BCD digit, the AF flag indicates a carry or borrow. The SF flag indicates the sign of a signed integer. The ZF flag indicates either a signed- or an unsigned-integer zero.

When performing multiple-precision arithmetic on integers, the CF flag is used in conjunction with the add with carry (ADC) and subtract with borrow (SBB) instructions to propagate a carry or borrow from one computation to the next.

The condition instructions Jcc (jump on condition code *cc*), SETcc (byte set on condition code *cc*), LOOPcc, and CMOVcc (conditional move) use one or more of the status flags as condition codes and test them for branch, set-byte, or end-loop conditions.

3.6.3.2. DF FLAG

The direction flag (DF, located in bit 10 of the EFLAGS register) controls the string instructions (MOVS, CMPS, SCAS, LODS, and STOS). Setting the DF flag causes the string instructions to auto-decrement (that is, to process strings from high addresses to low addresses). Clearing the DF flag causes the string instructions to auto-increment (process strings from low addresses to high addresses).

The STD and CLD instructions set and clear the DF flag, respectively.

The CWD instruction copies the sign (bit 15) of the word in the AX register into every bit position in the DX register. The CDQ instruction copies the sign (bit 31) of the doubleword in the EAX register into every bit position in the EDX register. The CWD instruction can be used to produce a doubleword dividend from a word before a word division, and the CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword division.

6.3.2.3. MOVE AND CONVERT

The MOVSX (move with sign extension) and MOVZX (move with zero extension) instructions move the source operand into a register then perform the sign extension.

The MOVSX instruction extends an 8-bit value to a 16-bit value or an 8- or 16-bit value to 32-bit value by sign extending the source operand, as shown in Figure 6-5. The MOVZX instruction extends an 8-bit value to a 16-bit value or an 8- or 16-bit value to 32-bit value by zero extending the source operand.

6.4. BINARY ARITHMETIC INSTRUCTIONS

The binary arithmetic instructions operate on 8-, 16-, and 32-bit numeric data encoded as signed or unsigned binary integers. Operations include the add, subtract, multiply, and divide as well as increment, decrement, compare, and change sign (negate). The binary arithmetic instructions may also be used in algorithms that operate on decimal (BCD) values.

6.4.1. Addition and Subtraction Instructions

The ADD (add integers), ADC (add integers with carry), SUB (subtract integers), and SBB (subtract integers with borrow) instructions perform addition and subtraction operations on signed or unsigned integer operands.

The ADD instruction computes the sum of two integer operands.

The ADC instruction computes the sum of two integer operands, plus 1 if the CF flag is set. This instruction is used to propagate a carry when adding numbers in stages.

The SUB instruction computes the difference of two integer operands.

The SBB instruction computes the difference of two integer operands, minus 1 if the CF flag is set. This instruction is used to propagate a borrow when subtracting numbers in stages.

6.4.2. Increment and Decrement Instructions

The INC (increment) and DEC (decrement) instructions add 1 to or subtract 1 from an unsigned integer operand, respectively. A primary use of these instructions is for implementing counters.

6.4.3. Comparison and Sign Change Instruction

The CMP (compare) instruction computes the difference between two integer operands and updates the OF, SF, ZF, AF, PF, and CF flags according to the result. The source operands are not modified, nor is the result saved. The CMP instruction is commonly used in conjunction with a Jcc (jump) or SETcc (byte set on condition) instruction, with the latter instructions performing an action based on the result of a CMP instruction.

The NEG (negate) instruction subtracts a signed integer operand from zero. The effect of the NEG instruction is to change the sign of a two's complement operand while keeping its magnitude.

6.4.4. Multiplication and Divide Instructions

The processor provides two multiply instructions, MUL (unsigned multiply) and IMUL signed multiply), and two divide instructions, DIV (unsigned divide) and IDIV (signed divide).

The MUL instruction multiplies two unsigned integer operands. The result is computed to twice the size of the source operands (for example, if word operands are being multiplied, the result is a doubleword).

The IMUL instruction multiplies two signed integer operands. The result is computed to twice the size of the source operands; however, in some cases the result is truncated to the size of the source operands (see “IMUL—Signed Multiply” in Chapter 3 of the *Intel Architecture Software Developer's Manual, Volume 2*).

The DIV instruction divides one unsigned operand by another unsigned operand and returns a quotient and a remainder.

The IDIV instruction is identical to the DIV instruction, except that IDIV performs a signed division.

6.5. DECIMAL ARITHMETIC INSTRUCTIONS

Decimal arithmetic can be performed by combining the binary arithmetic instructions ADD, SUB, MUL, and DIV (discussed in Section 6.4., “Binary Arithmetic Instructions”) with the decimal arithmetic instructions. The decimal arithmetic instructions are provided to carry out the following operations:

- To adjust the results of a previous binary arithmetic operation to produce a valid BCD result.
- To adjust the operands of a subsequent binary arithmetic operation so that the operation will produce a valid BCD result.

These instructions operate only on both packed and unpacked BCD values.

6.8.4. Test Instruction

The TEST instruction performs a logical AND of two operands and sets the SF, ZF, and PF flags according to the results. The flags can then be tested by the conditional jump or loop instructions or the SETcc instructions. The TEST instruction differs from the AND instruction in that it does not alter either of the operands.

6.9. CONTROL TRANSFER INSTRUCTIONS

The processor provides both conditional and unconditional control transfer instructions to direct the flow of program execution. Conditional transfers are taken only for specified states of the status flags in the EFLAGS register. Unconditional control transfers are always executed.

6.9.1. Unconditional Transfer Instructions

The JMP, CALL, RET, INT, and IRET instructions transfer program control to another location (destination address) in the instruction stream. The destination can be within the same code segment (near transfer) or in a different code segment (far transfer).

6.9.1.1. JUMP INSTRUCTION

The JMP (jump) instruction unconditionally transfers program control to a destination instruction. The transfer is one-way; that is, a return address is not saved. A destination operand specifies the address (the instruction pointer) of the destination instruction. The address can be a **relative address** or an **absolute address**.

A relative address is a displacement (offset) with respect to the address in the EIP register. The destination address (a near pointer) is formed by adding the displacement to the address in the EIP register. The displacement is specified with a signed integer, allowing jumps either forward or backward in the instruction stream.

An absolute address is an offset from address 0 of a segment. It can be specified in either of the following ways:

- **An address in a general-purpose register.** This address is treated as a near pointer, which is copied into the EIP register. Program execution then continues at the new address within the current code segment.
- **An address specified using the standard addressing modes of the processor.** Here, the address can be a near pointer or a far pointer. If the address is for a near pointer, the address is translated into an offset and copied into the EIP register. If the address is for a far pointer, the address is translated into a segment selector (which is copied into the CS register) and an offset (which is copied into the EIP register).

In protected mode, the JMP instruction also allows jumps to a call gate, a task gate, and a task-state segment.

6.9.1.2. CALL AND RETURN INSTRUCTIONS

The CALL (call procedure) and RET (return from procedure) instructions allow a jump from one procedure (or subroutine) to another and a subsequent jump back (return) to the calling procedure.

The CALL instruction transfers program control from the current (or calling procedure) to another procedure (the called procedure). To allow a subsequent return to the calling procedure, the CALL instruction saves the current contents of the EIP register on the stack before jumping to the called procedure. The EIP register (prior to transferring program control) contains the address of the instruction following the CALL instruction. When this address is pushed on the stack, it is referred to as the **return instruction pointer** or **return address**.

The address of the called procedure (the address of the first instruction in the procedure being jumped to) is specified in a CALL instruction the same way as it is in a JMP instruction (see Section 6.9.1.1., “Jump Instruction”). The address can be specified as a relative address or an absolute address. If an absolute address is specified, it can be either a near or a far pointer.

The RET instruction transfers program control from the procedure currently being executed (the called procedure) back to the procedure that called it (the calling procedure). Transfer of control is accomplished by copying the return instruction pointer from the stack into the EIP register. Program execution then continues with the instruction pointed to by the EIP register.

The RET instruction has an optional operand, the value of which is added to the contents of the ESP register as part of the return operation. This operand allows the stack pointer to be incremented to remove parameters from the stack that were pushed on the stack by the calling procedure.

See Section 4.3., “Calling Procedures Using CALL and RET”, for more information on the mechanics of making procedure calls with the CALL and RET instructions.

6.9.1.3. RETURN FROM INTERRUPT INSTRUCTION

When the processor services an interrupt, it performs an implicit call to an interrupt-handling procedure. The IRET (return from interrupt) instruction returns program control from an interrupt handler to the interrupted procedure (that is, the procedure that was executing when the interrupt occurred). The IRET instruction performs a similar operation to the RET instruction (see Section 6.9.1.2., “Call and Return Instructions”) except that it also restores the EFLAGS register from the stack. The contents of the EFLAGS register are automatically stored on the stack along with the return instruction pointer when the processor services an interrupt.

6.9.2. Conditional Transfer Instructions

The conditional transfer instructions execute jumps or loops that transfer program control to another instruction in the instruction stream if specified conditions are met. The conditions for control transfer are specified with a set of condition codes that define various states of the status flags (CF, ZF, OF, PF, and SF) in the EFLAGS register.

6.9.2.1. CONDITIONAL JUMP INSTRUCTIONS

The *Jcc* (conditional) jump instructions transfer program control to a destination instruction if the conditions specified with the condition code (*cc*) associated with the instruction are satisfied (see Table 6-4). If the condition is not satisfied, execution continues with the instruction following the *Jcc* instruction. As with the *JMP* instruction, the transfer is one-way; that is, a return address is not saved.

Table 6-4. Conditional Jump Instructions

Instruction Mnemonic	Condition (Flag States)	Description
Unsigned Conditional Jumps		
JA/JNBE	(CF or ZF)=0	Above/not below or equal
JAE/JNB	CF=0	Above or equal/not below
JB/JNAE	CF=1	Below/not above or equal
JBE/JNA	(CF or ZF)=1	Below or equal/not above
JC	CF=1	Carry
JE/JZ	ZF=1	Equal/zero
JNC	CF=0	Not carry
JNE/JNZ	ZF=0	Not equal/not zero
JNP/JPO	PF=0	Not parity/parity odd
JP/JPE	PF=1	Parity/parity even
JCXZ	CX=0	Register CX is zero
JECXZ	ECX=0	Register ECX is zero
Signed Conditional Jumps		
JG/JNLE	((SF xor OF) or ZF) = 0	Greater/not less or equal
JGE/JNL	(SF xor OF)=0	Greater or equal/not less
JL/JNGE	(SF xor OF)=1	Less/not greater or equal
JLE/JNG	((SF xor OF) or ZF)=1	Less or equal/not greater
JNO	OF=0	Not overflow
JNS	SF=0	Not sign (non-negative)
JO	OF=1	Overflow
JS	SF=1	Sign (negative)

The destination operand specifies a relative address (a signed offset with respect to the address in the EIP register) that points to an instruction in the current code segment. The *Jcc* instructions do not support far transfers; however, far transfers can be accomplished with a combination of a *Jcc* and a *JMP* instruction (see “*Jcc*—Jump if Condition Is Met” in Chapter 3 of the *Intel Architecture Software Developer's Manual, Volume 2*).

Table 6-4 shows the mnemonics for the *Jcc* instructions and the conditions being tested for each instruction. The condition code mnemonics are appended to the letter “J” to form the mnemonic for a *Jcc* instruction. The instructions are divided into two groups: unsigned and signed conditional jumps. These groups correspond to the results of operations performed on unsigned and signed integers, respectively. Those instructions listed as pairs (for example, *JA/JNBE*) are alternate names for the same instruction. The assembler provides these alternate names to make it easier to read program listings.

The *JCXZ* and *JECXZ* instructions test the *CX* and *ECX* registers, respectively, instead of one or more status flags. See Section 6.9.2.3., “Jump If Zero Instructions” for more information about these instructions.

6.9.2.2. LOOP INSTRUCTIONS

The *LOOP*, *LOOPE* (loop while equal), *LOOPZ* (loop while zero), *LOOPNE* (loop while not equal), and *LOOPNZ* (loop while not zero) instructions are conditional jump instructions that use the value of the *ECX* register as a count for the number of times to execute a loop. All the loop instructions decrement the count in the *ECX* register each time they are executed and terminate a loop when zero is reached. The *LOOPE*, *LOOPZ*, *LOOPNE*, and *LOOPNZ* instructions also accept the *ZF* flag as a condition for terminating the loop before the count reaches zero.

The *LOOP* instruction decrements the contents of the *ECX* register (or the *CX* register, if the address-size attribute is 16), then tests the register for the loop-termination condition. If the count in the *ECX* register is non-zero, program control is transferred to the instruction address specified by the destination operand. The destination operand is a relative address (that is, an offset relative to the contents of the *EIP* register), and it generally points to the first instruction in the block of code that is to be executed in the loop. When the count in the *ECX* register reaches zero, program control is transferred to the instruction immediately following the *LOOP* instruction, which terminates the loop. If the count in the *ECX* register is zero when the *LOOP* instruction is first executed, the register is pre-decremented to *FFFFFFFFH*, causing the loop to be executed 2^{32} times.

The *LOOPE* and *LOOPZ* instructions perform the same operation (they are mnemonics for the same instruction). These instructions operate the same as the *LOOP* instruction, except that they also test the *ZF* flag. If the count in the *ECX* register is not zero and the *ZF* flag is set, program control is transferred to the destination operand. When the count reaches zero or the *ZF* flag is clear, the loop is terminated by transferring program control to the instruction immediately following the *LOOPE/LOOPZ* instruction.

The *LOOPNE* and *LOOPNZ* instructions (mnemonics for the same instruction) operate the same as the *LOOPE/LOOPZ* instructions, except that they terminate the loop if the *ZF* flag is set.

6.9.2.3. JUMP IF ZERO INSTRUCTIONS

The *JECXZ* (jump if *ECX* zero) instruction jumps to the location specified in the destination operand if the *ECX* register contains the value zero. This instruction can be used in combination with a loop instruction (*LOOP*, *LOOPE*, *LOOPZ*, *LOOPNE*, or *LOOPNZ*) to test the *ECX* register prior to beginning a loop. As described in Section 6.9.2.2., “Loop Instructions”, the loop

Intel Architecture Software Developer's Manual

Volume 2: Instruction Set Reference

NOTE: The *Intel Architecture Software Developer's Manual* consists of three volumes: *Basic Architecture*, Order Number 243190; *Instruction Set Reference*, Order Number 243191; and the *System Programming Guide*, Order Number 243192.

Please refer to all three volumes when evaluating your design needs.

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Intel's Intel Architecture processors (e.g., Pentium® processor, Pentium processor with MMX™ technology, Pentium Pro processor, and Pentium II processor) may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation
P.O. Box 7641
Mt. Prospect IL 60056-7641

or call 1-800-879-4683
or visit Intel's website at <http://www.intel.com>

Copyright © Intel Corporation 1996, 1997.

* Third-party brands and names are the property of their respective owners.

ADD—Add

Opcode	Instruction	Description
04 <i>ib</i>	ADD AL, <i>imm8</i>	Add <i>imm8</i> to AL
05 <i>iw</i>	ADD AX, <i>imm16</i>	Add <i>imm16</i> to AX
05 <i>id</i>	ADD EAX, <i>imm32</i>	Add <i>imm32</i> to EAX
80 /0 <i>ib</i>	ADD r/m8, <i>imm8</i>	Add <i>imm8</i> to r/m8
81 /0 <i>iw</i>	ADD r/m16, <i>imm16</i>	Add <i>imm16</i> to r/m16
81 /0 <i>id</i>	ADD r/m32, <i>imm32</i>	Add <i>imm32</i> to r/m32
83 /0 <i>ib</i>	ADD r/m16, <i>imm8</i>	Add sign-extended <i>imm8</i> to r/m16
83 /0 <i>ib</i>	ADD r/m32, <i>imm8</i>	Add sign-extended <i>imm8</i> to r/m32
00 /r	ADD r/m8,r8	Add r8 to r/m8
01 /r	ADD r/m16,r16	Add r16 to r/m16
01 /r	ADD r/m32,r32	Add r32 to r/m32
02 /r	ADD r8,r/m8	Add r/m8 to r8
03 /r	ADD r16,r/m16	Add r/m16 to r16
03 /r	ADD r32,r/m32	Add r/m32 to r32

Description

Adds the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a carry in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

Operation

DEST ← DEST + SRC;

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

Protected Mode Exceptions

- #GP(0) If the destination is located in a nonwritable segment.
- If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

ADD—Add (Continued)

#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

Jcc—Jump if Condition Is Met

Opcode	Instruction	Description
77 <i>cb</i>	JA <i>rel8</i>	Jump short if above (CF=0 and ZF=0)
73 <i>cb</i>	JAE <i>rel8</i>	Jump short if above or equal (CF=0)
72 <i>cb</i>	JB <i>rel8</i>	Jump short if below (CF=1)
76 <i>cb</i>	JBE <i>rel8</i>	Jump short if below or equal (CF=1 or ZF=1)
72 <i>cb</i>	JC <i>rel8</i>	Jump short if carry (CF=1)
E3 <i>cb</i>	JCXZ <i>rel8</i>	Jump short if CX register is 0
E3 <i>cb</i>	JECXZ <i>rel8</i>	Jump short if ECX register is 0
74 <i>cb</i>	JE <i>rel8</i>	Jump short if equal (ZF=1)
7F <i>cb</i>	JG <i>rel8</i>	Jump short if greater (ZF=0 and SF=OF)
7D <i>cb</i>	JGE <i>rel8</i>	Jump short if greater or equal (SF=OF)
7C <i>cb</i>	JL <i>rel8</i>	Jump short if less (SF<>OF)
7E <i>cb</i>	JLE <i>rel8</i>	Jump short if less or equal (ZF=1 or SF<>OF)
76 <i>cb</i>	JNA <i>rel8</i>	Jump short if not above (CF=1 or ZF=1)
72 <i>cb</i>	JNAE <i>rel8</i>	Jump short if not above or equal (CF=1)
73 <i>cb</i>	JNB <i>rel8</i>	Jump short if not below (CF=0)
77 <i>cb</i>	JNBE <i>rel8</i>	Jump short if not below or equal (CF=0 and ZF=0)
73 <i>cb</i>	JNC <i>rel8</i>	Jump short if not carry (CF=0)
75 <i>cb</i>	JNE <i>rel8</i>	Jump short if not equal (ZF=0)
7E <i>cb</i>	JNG <i>rel8</i>	Jump short if not greater (ZF=1 or SF<>OF)
7C <i>cb</i>	JNGE <i>rel8</i>	Jump short if not greater or equal (SF<>OF)
7D <i>cb</i>	JNL <i>rel8</i>	Jump short if not less (SF=OF)
7F <i>cb</i>	JNLE <i>rel8</i>	Jump short if not less or equal (ZF=0 and SF=OF)
71 <i>cb</i>	JNO <i>rel8</i>	Jump short if not overflow (OF=0)
7B <i>cb</i>	JNP <i>rel8</i>	Jump short if not parity (PF=0)
79 <i>cb</i>	JNS <i>rel8</i>	Jump short if not sign (SF=0)
75 <i>cb</i>	JNZ <i>rel8</i>	Jump short if not zero (ZF=0)
70 <i>cb</i>	JO <i>rel8</i>	Jump short if overflow (OF=1)
7A <i>cb</i>	JP <i>rel8</i>	Jump short if parity (PF=1)
7A <i>cb</i>	JPE <i>rel8</i>	Jump short if parity even (PF=1)
7B <i>cb</i>	JPO <i>rel8</i>	Jump short if parity odd (PF=0)
78 <i>cb</i>	JS <i>rel8</i>	Jump short if sign (SF=1)
74 <i>cb</i>	JZ <i>rel8</i>	Jump short if zero (ZF = 1)
0F 87 <i>cw/cd</i>	JA <i>rel16/32</i>	Jump near if above (CF=0 and ZF=0)
0F 83 <i>cw/cd</i>	JAE <i>rel16/32</i>	Jump near if above or equal (CF=0)
0F 82 <i>cw/cd</i>	JB <i>rel16/32</i>	Jump near if below (CF=1)
0F 86 <i>cw/cd</i>	JBE <i>rel16/32</i>	Jump near if below or equal (CF=1 or ZF=1)
0F 82 <i>cw/cd</i>	JC <i>rel16/32</i>	Jump near if carry (CF=1)
0F 84 <i>cw/cd</i>	JE <i>rel16/32</i>	Jump near if equal (ZF=1)
0F 84 <i>cw/cd</i>	JZ <i>rel16/32</i>	Jump near if 0 (ZF=1)
0F 8F <i>cw/cd</i>	JG <i>rel16/32</i>	Jump near if greater (ZF=0 and SF=OF)

Jcc—Jump if Condition Is Met (Continued)

Opcode	Instruction	Description
0F 8D <i>cw/cd</i>	JGE <i>rel16/32</i>	Jump near if greater or equal (SF=OF)
0F 8C <i>cw/cd</i>	JL <i>rel16/32</i>	Jump near if less (SF<>OF)
0F 8E <i>cw/cd</i>	JLE <i>rel16/32</i>	Jump near if less or equal (ZF=1 or SF<>OF)
0F 86 <i>cw/cd</i>	JNA <i>rel16/32</i>	Jump near if not above (CF=1 or ZF=1)
0F 82 <i>cw/cd</i>	JNAE <i>rel16/32</i>	Jump near if not above or equal (CF=1)
0F 83 <i>cw/cd</i>	JNB <i>rel16/32</i>	Jump near if not below (CF=0)
0F 87 <i>cw/cd</i>	JNBE <i>rel16/32</i>	Jump near if not below or equal (CF=0 and ZF=0)
0F 83 <i>cw/cd</i>	JNC <i>rel16/32</i>	Jump near if not carry (CF=0)
0F 85 <i>cw/cd</i>	JNE <i>rel16/32</i>	Jump near if not equal (ZF=0)
0F 8E <i>cw/cd</i>	JNG <i>rel16/32</i>	Jump near if not greater (ZF=1 or SF<>OF)
0F 8C <i>cw/cd</i>	JNGE <i>rel16/32</i>	Jump near if not greater or equal (SF<>OF)
0F 8D <i>cw/cd</i>	JNL <i>rel16/32</i>	Jump near if not less (SF=OF)
0F 8F <i>cw/cd</i>	JNLE <i>rel16/32</i>	Jump near if not less or equal (ZF=0 and SF=OF)
0F 81 <i>cw/cd</i>	JNO <i>rel16/32</i>	Jump near if not overflow (OF=0)
0F 8B <i>cw/cd</i>	JNP <i>rel16/32</i>	Jump near if not parity (PF=0)
0F 89 <i>cw/cd</i>	JNS <i>rel16/32</i>	Jump near if not sign (SF=0)
0F 85 <i>cw/cd</i>	JNZ <i>rel16/32</i>	Jump near if not zero (ZF=0)
0F 80 <i>cw/cd</i>	JO <i>rel16/32</i>	Jump near if overflow (OF=1)
0F 8A <i>cw/cd</i>	JP <i>rel16/32</i>	Jump near if parity (PF=1)
0F 8A <i>cw/cd</i>	JPE <i>rel16/32</i>	Jump near if parity even (PF=1)
0F 8B <i>cw/cd</i>	JPO <i>rel16/32</i>	Jump near if parity odd (PF=0)
0F 88 <i>cw/cd</i>	JS <i>rel16/32</i>	Jump near if sign (SF=1)
0F 84 <i>cw/cd</i>	JZ <i>rel16/32</i>	Jump near if 0 (ZF=1)

Description

Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the flags are in the specified state (condition), performs a jump to the target instruction specified by the destination operand. A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the *Jcc* instruction.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). A relative offset (*rel8*, *rel16*, or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32-bit immediate value, which is added to the instruction pointer. Instruction coding is most efficient for offsets of -128 to +127. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s, resulting in a maximum instruction pointer size of 16 bits.

Jcc—Jump if Condition Is Met (Continued)

The conditions for each *Jcc* mnemonic are given in the “Description” column of the table on the preceding page. The terms “less” and “greater” are used for comparisons of signed integers and the terms “above” and “below” are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the *JA* (jump if above) instruction and the *JNBE* (jump if not below or equal) instruction are alternate mnemonics for the opcode 77H.

The *Jcc* instruction does not support far jumps (jumps to other code segments). When the target for the conditional jump is in a different segment, use the opposite condition from the condition being tested for the *Jcc* instruction, and then access the target with an unconditional far jump (*JMP* instruction) to the other segment. For example, the following conditional far jump is illegal:

```
JZ FARLABEL;
```

To accomplish this far jump, use the following two instructions:

```
JNZ BEYOND;
JMP FARLABEL;
BEYOND:
```

The *JECXZ* and *JCXZ* instructions differs from the other *Jcc* instructions because they do not check the status flags. Instead they check the contents of the *ECX* and *CX* registers, respectively, for 0. Either the *CX* or *ECX* register is chosen according to the address-size attribute. These instructions are useful at the beginning of a conditional loop that terminates with a conditional loop instruction (such as *LOOPNE*). They prevent entering the loop when the *ECX* or *CX* register is equal to 0, which would cause the loop to execute 2^{32} or 64K times, respectively, instead of zero times.

All conditional jumps are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

Operation

```
IF condition
    THEN
        EIP ← EIP + SignExtend(DEST);
        IF OperandSize = 16
            THEN
                EIP ← EIP AND 0000FFFFH;
        FI;
    FI;
```

Flags Affected

None.

Jcc—Jump if Condition Is Met (Continued)**Protected Mode Exceptions**

#GP(0) If the offset being jumped to is beyond the limits of the CS segment.

Real-Address Mode Exceptions

#GP If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if 32-address size override prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if 32-address size override prefix is used.